

Rally Software Development Corporation
Whitepaper

Tactical Management of Agile Development: *Achieving Competitive Advantage*

Dean Leffingwell
and
Dave Muirhead
062204

1655 Walnut Street, Suite 200
Boulder, Colorado 80302

v 303 226 1180
f 303 226 1179
www.rallydev.com

Table of Contents

Competitive Advantage in a Software Economy.....	3
Software Process – the Big Black Box.....	3
Achieving Visibility with Agile Methods.....	4
Overview of Agile Methods	4
Scrum.....	5
Extreme Programming (XP).....	6
Rational Unified Process (RUP).....	6
Applicability of Agile Methods.....	8
An Agile Method Scale.....	8
Scaling Up.....	8
The Cornerstone of Agility – Iterative Development	9
Characteristics of Iterative Development.....	9
Fixed Length Iterations (Time Boxing).....	9
Two-Level Planning	9
Commitment to the Iteration Plan.....	10
Just-In-Time Requirements Elaboration.....	10
Early and Continuous Testing	10
Continuous Learning and Adaptation	11
The Agile Development Process	11
Overview.....	11
Defining the System.....	12
Release Planning.....	13
Iteration Planning and Execution.....	17
Small Releases.....	20
Tracking and Adjusting.....	20
Conclusion	22
Suggested Reading.....	23
Bibliography	23

Competitive Advantage in a Software Economy

The software product development industry has evolved to become one of the most important industries of our time. Employing millions of practitioners throughout virtually every developed country worldwide, this industry creates some of the most essential products we use to maintain and extend our lifestyles. From controlling the production of the food we eat, to providing safety and control of the vehicles we drive, to providing life sustaining medical advances and automating the businesses that employ us, software has become the embodiment of much of the world's most valuable intellectual property.

In this intensely competitive environment, what will separate the winners from the losers, the leaders from the second place finishers? Simply, it is their *ability to more quickly create and deliver software products that better address their customer's real needs*. Successful companies can be characterized by:

- 1) they are often first to market
- 2) their solutions directly address customer's real pain points, and they have built in mechanisms to assure that they do so
- 3) they adapt more rapidly to business and technological change than their competitors

*In other words, they **deliver early and they deliver often**, and they are relentless in constantly enhancing their solution to assure an ongoing fit to their customers needs.*

This mantra seems simple enough, so why doesn't everybody do it? The answer lies deep inside the software development process itself, and we have observed that those who master this demanding and difficult process are most likely to emerge as the winners.

Software Process – the Big Black Box

From the perspective of customers or other stakeholders, and even software managers and executives, the software process too often looks like a big black box.

Requirements go in, and a product comes out later, typically much later. In the meantime we may see requirements specifications, design models and other work products which assure us something is happening, but we cannot be sure what. Moreover, the probability of requirements change increases with the length of the development cycle, so we increasingly doubt that the emerging product will meet our needs.



Figure 1 - Software Process Black Box

So we wait and hope. However, the results are at best uncertain and often disappointing. Even under the auspices of a reasonably effective process, when seeing the product for the very first time, we are likely to get the “Yes, But” syndrome reaction from our customers [Leffingwell 2003]:

“YES, this is really nice and we appreciate what you’ve done.”

Followed by:

“BUT no, it’s not exactly what we need, perhaps

- a) you didn’t understand what we meant
- b) you did understand but you did something different anyway, or
- c) it is what we meant and what we said but now we need something different.”

What we need is a way for customers to see and evaluate progress *before* it is too late. We need a process that is more *agile* and more *responsive*. One that gives key stakeholders *visibility* early and often-*visibility* into the solution in its current state, and *visibility* into the process used to create the solution. We need a process that accepts change while also having sufficient rigor to yield a quality solution.

Achieving Visibility with Agile Methods

Fortunately, we can address this dilemma by implementing software practices that are more adaptive, more agile and most importantly, more *visible* to the key stakeholders who depend on the outcome. Into this void come a number of new software practices, an industry movement towards more agile and adaptive software development. While they differ in name, presentation, terminology and scope, they all focus on two common objectives:

If we can see early, we can know earlier. If we know earlier, we can adapt.
--

1. Develop usable software more quickly
2. Provide timely and regular visibility of the solution to customers, product owners and other key stakeholders

“Agility, for a software development organization, is the ability to adapt and react expeditiously and appropriately to changes in its environment and to demands imposed by this environment...An agile process is one that readily embraces and supports this degree of adaptability. So, it is not simply about the size of the process or the speed of delivery; it is mainly about flexibility.” [Kruchten 2001]

One common aspect of all these methods is that each provides visibility through the presence of *iterations*.

An iteration is a sequence of development activities conducted according to a plan and evaluation criteria that culminates in the delivery of a self-consistent, integrated and tested increment of software.

Iterations have the express purpose of providing, in a short period of time, objective evidence of the functionality, quality and fitness for use of a software application under development.

With a process that develops via iterations, we have near term visibility. We have objective evidence. We have immediate feedback on both the product and process, and we have it early enough to give us many options, rather than none or just a few. Surely a process that produces observable and measurable increments of software frequently has a competitive advantage over any process that does not. In the next section, we'll look at some iterative and agile development practices and see if they can deliver on the *vision of visibility*.

Overview of Agile Methods

As with most software disciplines, we eventually agree on the major issues, but when it comes to implementation the practitioner must often pick and choose amongst practices that are somewhat alike and yet seemingly very different at the same time. Such is the case with the movement to agile methods. Practitioners have (too) many to choose from.

In this section we'll describe three popular agile methods and see what they have in common. In later sections, we'll distill these into a set of common practices that create a method-independent framework which can be applied by most software teams. In addition, we'll discover that scaling these methods requires new approaches and tooling to facilitate the definition, development and delivery of these applications.

Scrum

Scrum (Figure 2) is an agile project management method that is enjoying increasing popularity and effective use. Jeff Sutherland and Ken Schwaber [Schwaber 2002] at Easel developed many of the original Scrum practices in 1993. Thereafter, Scrum was formalized and subsequently presented at OOPSLA'96. Since then, Sutherland, Schwaber and others have extended and enhanced it at many software companies and IT organizations.

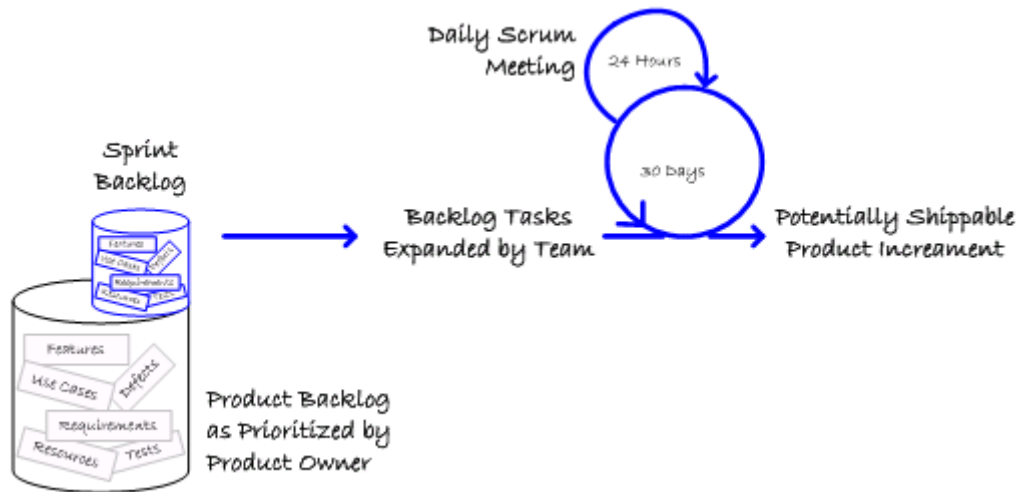


Figure 2 - Scrum Process Model

Key Scrum practices include:

- *Sprints* are iterations of a fixed 30 days duration
- Work within a sprint is fixed. Once the scope of a Sprint is committed, no additional functionality can be added except by the development team
- All work to be done is characterized as *product backlog* which includes requirements to be delivered, defect workload, as well as infrastructure and design activities.
- A Scrum Master mentors and manages the self organizing and self accountable teams that are responsible for delivery of successful outcomes at each sprint
- A daily *standup* meeting is a primary communication method
- A heavy focus on *time boxing*. Sprints, standup meetings, release review meetings and the like are all completed in prescribed times.
- Scrum also allows requirements, architecture and design to emerge over the course of the project.

Scrum, like RUP talks in terms of lifecycle phases In the case of Scrum, the phases are Planning, Staging, Development, and Release.

There are typically a small number of development sprints to reach a release. Later sprints focus more on system level quality and performance as well as documentation and other activities necessary to support a deployed product.

Typical Scrum guidance calls for fixed 30 day sprints, with approximately three sprints per release, thus supporting incremental market releases on a 90 day timeframe.

Extreme Programming (XP)

Extreme Programming (Figure 3) is a widely used agile method described in a number of books by Kent Beck [Beck 2000] and others. Key practices of XP include:

- A team of five to ten programmers work at one location with customer representation on site.
- Development occurs in frequent builds or iterations, each of which is releasable and delivers incremental functionality.
- Requirements are specified as *user stories*, each a chunk of new functionality the user requires.
- Programmers work in pairs, follow strict coding standards and do their own unit testing
- Requirements, architecture and design emerge over the course of the project.

XP is prescriptive in scope, and the authors note that it is best applied to small teams of under 10 developers where a customer is either integral to the team or readily accessible. In addition, the “P” in XP stands for “programming” and, as opposed to the other methods. XP describes some innovative and often controversial practices for the actual writing of software.

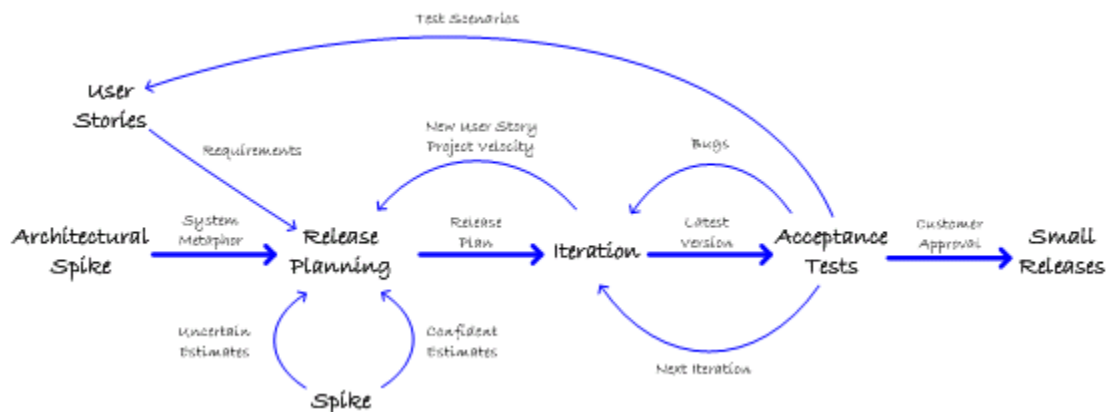


Figure 3 - XP Process Model

With its programming perspective, XP focuses less on the overall software lifecycle and more on a software release. Its heaviest emphasis is on the near term iterations, which are set at 1-3 weeks in length.

Rational Unified Process (RUP)

The Rational Unified Process [Rational 2002] is a popular process framework characterized in numerous books and is also available as a commercial product marketed by IBM's Rational Software.

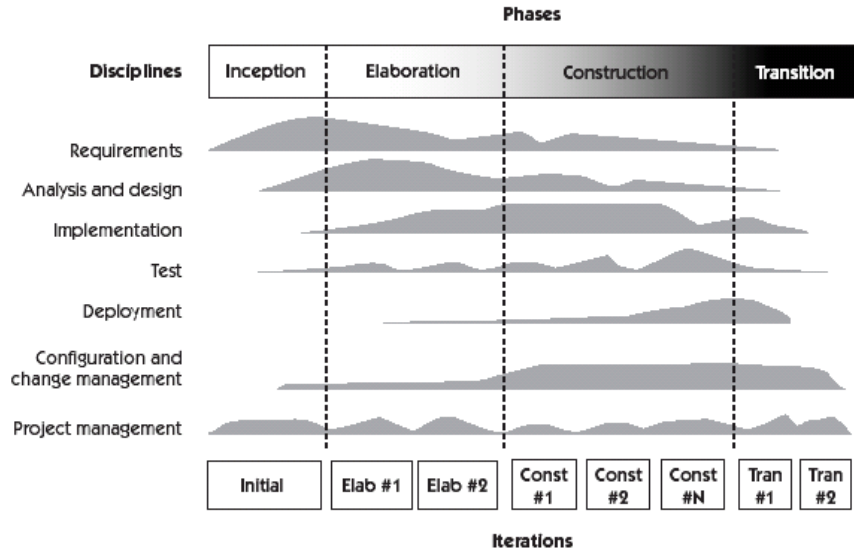


Figure 4 - RUP Process Model

The RUP was developed in the late 1990s in conjunction with the development of the Unified Modeling Language. Contributors included the UML authors Grady Booch, Ivar Jacobson, Jim Rumbaugh, as well as a number of other industry leaders, including Philippe Kruchten and Walker Royce. The RUP leverages the UML heavily in the requirements and architectural activities.

The RUP provides a full lifecycle approach covering a series of natural product lifecycle phases labeled as Inception, Elaboration, Construction and Transition. Within each phase, the project typically undergoes multiple iterations. Each iteration builds on the functionality of the prior iteration; thus, the project is developed in an “iterative and incremental” fashion, whereby each new iteration adds to the functionality of the prior iteration. The software application evolves in this fashion with the benefit of regular and continuous feedback.

The actual number and length of iterations are not prescribed as they are based heavily on the team size and criticality of the application, but guidance points to iterations of from 2-6 weeks in length.

Applicability of Agile Methods

We've described the migration to agile methods as an industry "movement". As with any movement, the trend is not without its critics, and even agile advocates vociferously debate the effectiveness of one method versus another. But on analysis it appears that most debates can be resolved by looking at method differences in the context of specific projects. A method that works in one team may work less well in another. A team of five people in a room with a customer that is creating a single hosted application, and a team of 900 developers in nine countries synchronizing a product release that ships to 20,000 customers worldwide are unlike things. How could we ever assume these teams could effectively apply the same software development practices? Indeed they cannot.

With respect to agile methods, the analysis centers primarily on project scale and criticality. With respect to scale, we all agree that constant informal communication of requirements between customers and the team is good, but as the team scales and the customers become more numerous and more remote, informal communication becomes impractical and other methods must be applied. Cockburn highlights this point with a discussion of needed methodology "weight" versus problem size and number of people required to succeed. (Figure 5). Bigger teams building larger applications require heavier methodologies. Heavier methodologies provide better structure but are harder to change and are therefore less agile.

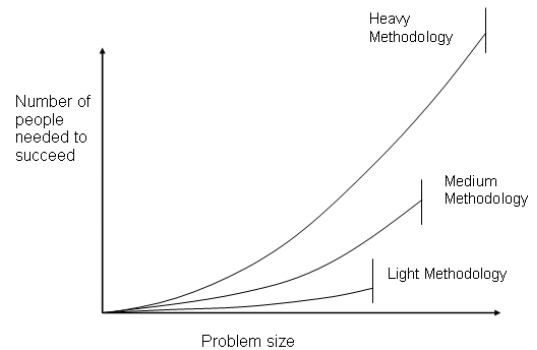


Figure 5 - Cockburn Weight Scale

Further complicating the discussion is the issue of application criticality. A small team building a pacemaker program or a medium team building a securities trading application will need heavier software practices than a team prototyping a personal aggregation portal.

In addition, team *location* is highly relevant to the discussion. If a change can be communicated verbally in seconds in a common language to people gathered in a room, that's straightforward. If two team members (or 200!) work in different locations and different time zones the same change has extraordinarily different impact. To be successful, the software methods we deploy must take these factors into account.

Perhaps we can still be agile, but we need to be agile differently.

An Agile Method Scale

Summarizing these factors and the methods we describe, we arrive at the chart in Figure 6. On the lower left of the scale we have the most agile applications and the most agile applicable team sizes. Moving to the upper right, we find larger and more critical applications and we require heavier weight methodologies which are more robust and therefore less agile.

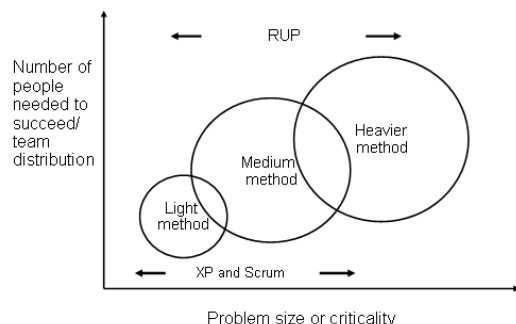


Figure 6 - Agile Methodology Weight Scale

Scaling Up

Clearly, given the improved time to market and visibility of agile practices, we'll want to leverage them no matter how far up and to the right our project context takes us. Fortunately, this is feasible if we adapt some of the practices and apply a little tooling to the challenge. Specifically, we'll need to:

- 1) Adopt requirements practices that scale to the communication and specificity challenges of larger and more critical practices

- 2) Leverage a shared repository of requirements, defects, test cases and other work products so we can more effectively communicate and track our progress in our larger and more distributed team
- 3) Automate some of the more intense, tactical management challenges of iterative development
- 4) Automate some of the real time testing practices to facilitate testing rapid iterations.
- 5) Leverage the power of the net to facilitate constant real time communication and collaboration on an ad hoc basis

We'll look at some of these differing practices as we further analyze and apply these agile methods.

The Cornerstone of Agility – Iterative Development

Although they describe themselves differently, Scrum, XP and RUP share many values, practices and techniques. In each, the cornerstone practice is the proven strategy of iterative and incremental development, or simply *iterative development*.

Iterative development divides the development of software releases into a series of *iterations*. Each iteration is a short period during which a subset of the system's requirements is elaborated, designed, coded and tested, culminating in the delivery of an integrated, useful *increment* of software. A key feature of true iterative development is that every iteration including the first iteration delivers *production-quality* code, not just requirement specifications, design artifacts or a prototype.

Incremental development divides the planned capabilities of a system into cohesive, valuable subsets, and then develops and delivers one such subset per *iteration*. This strategy results in the delivery of a running, useful, stable and potentially production deployable *increment* of the final production system after every iteration.

The primary measure of progress in an iterative and incremental process – in fact the *only thing* that the team gets credit for – is the delivery of working software accompanied by tests that verify its quality and adherence to requirements.

Iterative process models deliver a key benefit - they deliver working, tested software earlier and more often, resulting in greater visibility into the current state of the software product and into the process creating the software product.

Characteristics of Iterative Development

The iterative development strategy has a number of important characteristics.

Fixed Length Iterations (Time Boxing)

Most agile methods advise fixing the duration and end date of iterations. The usual guidance is that iterations should be 2-8 weeks in duration.

This time boxing establishes a rhythm for the development organization that becomes the drum beat that synchronizes the activities of all participants. Like a manufacturing schedule, it is repetitive, predictable and reliable and all software production and delivery rotates around this cycle. The most important benefit is that time boxing introduces a near-term milestone that forces both the team and code lines to converge and actually deliver working software at regular intervals.

Two-Level Planning

Agile and iterative development eschews predictive planning. As Kruchten, architect of the RUP and many large, complex software projects, reminds us:

“There have been countless ambitious but doomed attempts to plan large software projects from start to finish in minute detail, as you would plan the construction of a skyscraper...

In an iterative process, we recommend that the development be based on two kinds of plans:

- *A coarse-grained plan: the phase plan*
- *A series of fine-grained plans: the iteration plans” [Kruchten 2004]*

So, the investment in long-range (longer than one or two iterations), speculative plans is reduced and the lack of precision and accuracy in such plans is acknowledged. Release plans (what Kruchten refers to as phase plans) are therefore intentionally coarse-grained, less comprehensive and less precise.

Iterations plans, on the other hand, are focused on the short run, are less speculative and are worthy of greater estimating investment and precision.

In the end, however, all plans are subject to adjustment based on the latest facts and observations of reality.

Commitment to the Iteration Plan

Larman [Larman 2004] notes:

“Iterative and agile methods embrace change; but not chaos. In a sea of constant change, a point of stability is necessary.”

The development team can and should be expected to embrace change. But in order for it to fulfill its responsibility of delivering a potentially deployable increment of the product at the end of each iteration, the team must be able to commit to and deliver on the iteration plan. Changing the iteration plan midstream invites chaos and severely threatens the ability of the development team to deliver at iteration end.

It is important that the development team deliver working software at every iteration for several reasons: 1) It is the only way to objectively demonstrate progress, 2) It is the only way users can provide concrete feedback based on actual usage of the software, and 3) like a missed manufacturing deadline, it hurts the company's rhythm and morale when the team cannot deliver on an iteration. Moreover, the shorter the iteration, the more practical it becomes to freeze an iteration while maintaining the team's ability to embrace change.

Just-In-Time Requirements Elaboration

Another characteristic of iterative development is that the investment in elaborating requirements is deferred until the “last responsible moment”. When the “last responsible moment” actually occurs is project dependent and based on a number of factors such as the size, criticality and profile of the project, as well as the characteristics of the specific requirements being elaborated.

Release planning depends on *identifying* the most important capabilities and characteristics of the system, but it only requires that they be *elaborated* to an extent that it is possible to produce preliminary, low-confidence, order-of-magnitude estimates.

Iteration planning requires somewhat greater elaboration of requirements, but now only to the extent that a reasonably high-confidence estimate is produced.

Coding requires that the to-be-coded requirements be sufficiently defined such that they can be accurately translated into acceptance tests and running code.

Early and Continuous Testing

Early and continuous testing is a mandatory requirement of iterative development. Because the system is developed in short iterations that deliver cohesive, valuable increments of functionality, various kinds of testing that used to be deferred until “delivery” must now be started during the *first* iteration. This forces a

“test early and often” cycle which becomes integral to the iteration process. “Design then build then test later” becomes “designbuildtest” within the scope of each iteration.

The programmers and testers should be writing and executing tests as they write the code. By involving the test team immediately the whole project learns valuable lessons sooner, risks are discovered and dealt with earlier, and the quality of the software can be dramatically improved.

Continuous Learning and Adaptation

Iterative development processes are intentionally organized so that the entire team, including end users, has an opportunity to reflect on the results of the process often, to learn from that examination, and then to adapt the process to produce better results in the next iteration. At the end of each iteration it is important to reflect on what worked, what did not, and then make decisions about what to do differently next time. The shorter the iteration, the faster the learning and the virtuous cycle repeats.

The Agile Development Process

In this section, we discuss a generalized agile process based on the common elements of these popular methods and the cornerstone practice of iterative development. We'll review the primary activities in the process including *system definition*, *release planning*, *iteration planning and execution*, and *acceptance*, as well as the continuous underlying activity of *tracking and adjusting*.

Overview

Earlier we characterized the repeating cycle of time-boxed iterations as the drum beat that synchronizes the entire software development organization. There is another important cycle to consider: the *release cycle*. It is a cycle because there will usually be more than one release of the software product, and so we expect to go through the entire process of developing and delivering a release more than once. The release cycle is normally several iterations in length and encompasses the iteration cycle.

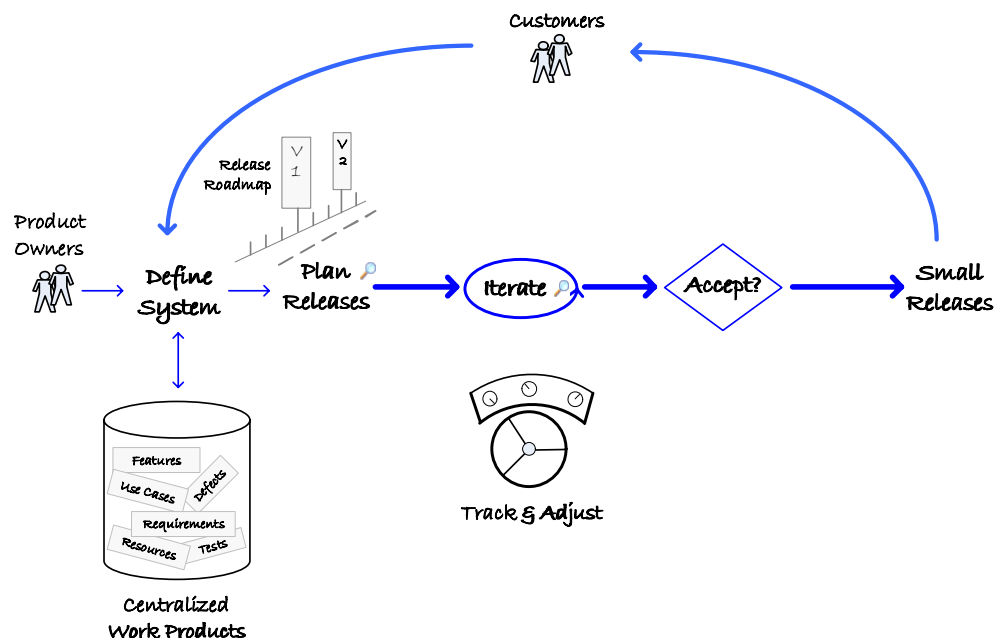


Figure 7 - Generalized Agile Process Model

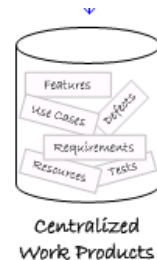
In our generalized agile process, each release cycle consists of a period of *system definition*, followed by *release planning*, followed by a series of *iterations* that each produce a new increment of capability, followed by the *delivery* of a new version of the system into production for the customer.

Defining the System

We've described "just in time requirements elaboration" as a key aspect of agile practices. However, this oversimplifies the case because "waiting until the last possible responsible moment" does not address questions such as when that moment might be, how dependent it is on the scope or impact of a specific requirement, what to do with future requirements that are suggested but not yet in an iteration plan and the like. In this section, we'll describe some agile requirements management practices and how to implement them within the context of the agile method described above.

Requirements Repository

To start, we note that system definition and requirements capture do not happen solely within the limited scope of a specific release plan. Rather, these activities happen within the overall application lifecycle. At any point in time we have likely collected a large number of "could do", "should do" and "must do" requirements. These should be aggregated in a centralized repository where they can be viewed prioritized, and "mined" for iteration features. In any case, they should be captured as they are discovered so they can be reasoned about at some later point in the process. Requirements should also be readily accessible to all team members, available to be enhanced and revised over time, and remain reasonably current to directly drive testing as they are implemented.

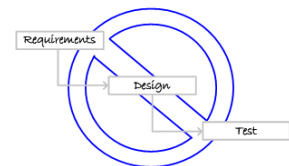


The expression of requirements may vary widely. They could come in the form of features ("automatic upgrade"), user stories, semantically rich use cases ("withdraw cash") or even defects. Some will be non-functional but specific in nature ("the agent will deliver 400 mbits/second bandwidth"). Others will be more intangible and state aspects of the system that are difficult to measure ("the system must be easy to install"). In some cases they come in a more formal specification or they may be contained elsewhere in a published standard. In general, agile teams will need to become comfortable with a variety of these forms and understand how to apply them in the agile context.

In all cases, enabling the team to easily capture requirements, search them, prioritize them and elaborate as necessary is the primary function of the repository.

Continuous Elaboration

One interesting aspect of the evolution from a more waterfall-like process to agile and iterative approaches is the subtle yet critical change to the requirements elaboration process. In the waterfall model, we liked to assume we would be given a reasonably complete set of requirements that could be implemented. This meant someone spent the time inventing them, defining them and vetting them with the key stakeholders. We also assumed that this time was invested before the development process began. It may or may not have worked, but at least it was easy to describe!



In agile practices requirements are more abstract. They are never frozen (except within the bounds of short iterations) and the team has much more accountability in making certain they are building the right thing. This means the agile team has a higher degree of involvement in defining the system under construction. In turn, this means the development team members and product owners work hand-in-hand in a nearly continuous process. Our design-build-test mantra evolves to elaborate-design-build-test. Elaboration becomes everyone's responsibility. In other words:

If you are not elaborating, you are not making progress

Breaking Big Features into Iteration Size Bites

Another characteristic of our departed waterfall model is that we assumed we had significant and meaningful time to implement real requirements; timeframes that could be measured in months, not days or weeks. In our agile methods we may have as little as 2 - 4 weeks to deliver the feature. How do we accomplish this? Well, nothing about agile practices changes the time it takes to write a line of code so we'll need to factor these larger features into bite size chunks. The team estimates and implements the larger feature over the course of multiple iterations. Each iteration yields some demonstrable value, perhaps at a minimum providing a proof-of-concept demonstration. With some practice, teams discover this produces better results overall as the eventual result is better aligned to real customer value. In the rare case where it is simply not feasible to break up the feature, the best practice is to branch the code and put that feature on a separate, yet synchronized code iteration plan.

Allocating Requirements to the Release Plan

The next step in the process is to allocate the requirements to the release plan by determining which release, and possibly which iteration, a requirement needs to be assigned. There is no one right way to do this, but considerations should include:

- Relative priority of this requirement with respect to others
- External market events such as competitive announcements, trade shows product themes, analyst briefings and the like
- Any necessary coupling or binding to other requirements that are planned
- Constraints and availability of critical team members skills

Release Planning

Release planning (Figure 6) is the set of scoping and scheduling activities conducted by product owners, the development team and often customers, at the beginning of each release cycle.

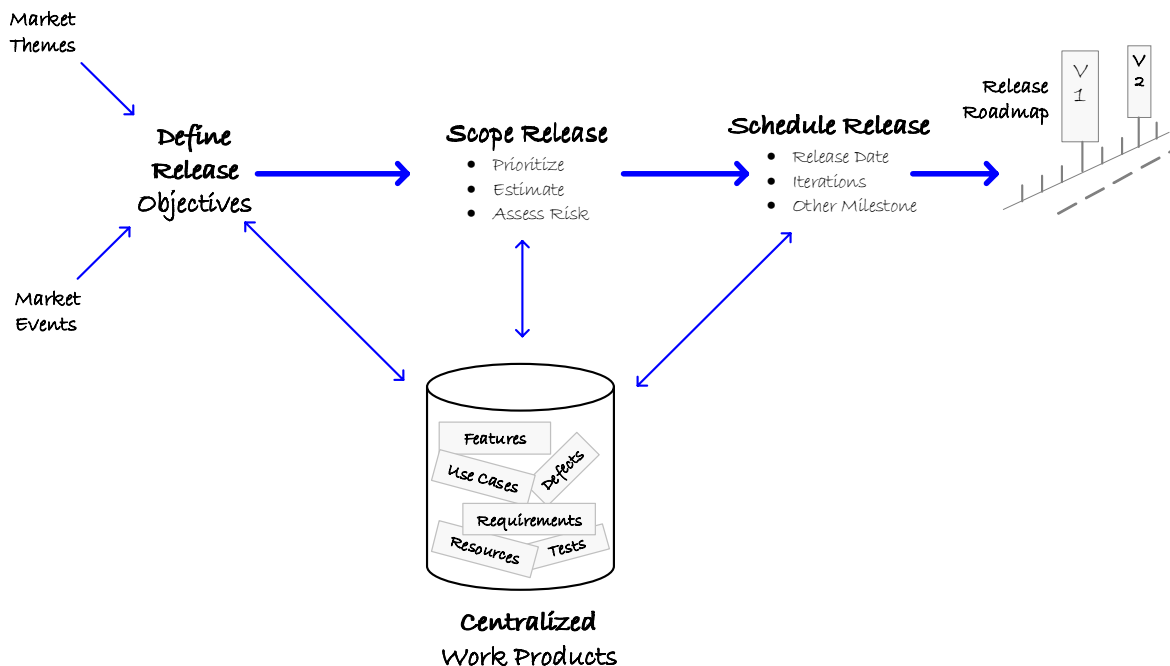


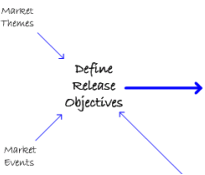
Figure 8 - Release Planning Process Model

The result of release planning is a *release plan* (or *release roadmap*, if multiple releases are planned) that:

1. identifies the features, defect fixes and other work that the team hopes to deliver in the release;
2. establishes the schedule for the release, including the duration and end dates of the iterations;
3. identifies the resources that will be available to develop the release and estimates their rate of production; and
4. defines the quality and readiness criteria that will be required of the new release.

It is understood that release plans are neither comprehensive nor guaranteed. Instead, they are updated as often as necessary to reflect changing business priorities and needs. Nevertheless, release plans serve to set goals and expectations internally and externally, and they guide and constrain the follow-on iteration planning activities.

Defining the Release Objectives



The first step in release planning is to *define the objectives* for the release. This analysis takes into account marketplace events, product or positioning and themes, competitive threats and opportunities, the needs of specific customers or partners, and any significant outstanding support cases. During this phase, product owners and customers have the opportunity to outline new functionality and refine existing plans for future functionality in order to better address current business realities and opportunities. The resulting release objectives form a framework for choosing the scope and schedule of the release.

Scoping the Release

Scope Release

- Prioritize
- Estimate
- Assess Risk

Software projects are best driven by either a fixed scope of functionality that must be delivered within a flexible timeframe, or else by a fixed schedule in which a flexible scope must be delivered. Quality and resources are not commonly used as planning and management levers. The decision to take a scope-driven or schedule-driven approach is complicated and must be made considering market pressures, opportunities and business constraints.

The process of *scoping* a release is conducted by the product owners and the development team and usually includes external customers.

Estimating

The development team is responsible for providing a rough order-of-magnitude *estimate* of the time to develop each feature, defect or other chunk of work that the product owners propose. The product owners and development team meet as necessary to discuss those items until the development team has enough information to make rough estimates.

The estimates created during release planning are for much larger chunks of work than estimates created during iteration planning. It is common that release planning estimates are weeks of effort in size, whereas iteration planning estimates will generally be one to three days.

Assessing Risk

The development team may be asked to *assess the risk* associate with each proposed feature. Risk usually includes factors such as the technical complexity of an item, whether the team has ever implemented a similar feature, how well understood the item is, how architecturally significant the item is, and so on.

Defining the Backlog

The product owners, with contributions from customers and the development team, are responsible for defining the set of features, defects and other work that could be included in the scope of an upcoming release. The Scrum methodology refers to this list as the *product backlog*.

Since it is generally impossible to deliver every item in the product backlog in one release, product owners are forced to choose which items from the product backlog to propose for the release. The subset of the product backlog that is chosen for a release is referred to as the *release backlog*.

One important aspect of the Scrum notion of backlogs is that their contents are ordered from highest business value to lowest business value. By definition, then, the release backlog contains the set of features, defects and other work that the business has identified as being most valuable to deliver in the next release.

Prioritizing The Backlog

As such, product owners must *prioritize* the items in the product backlog. The primary criteria for prioritizing a given backlog item is the value that would accrue to the business by delivering that item in the release.

To make this choice, the product owner must synthesize the release objectives, business value, risk and development cost information to establish a ranking of the items in the product backlog. This ranking indicates the order in which the product owner would like the development team to deliver the items in the product backlog.

Schedule-Driven or Scope-Driven?

If the project is scope-driven, the product owners can then choose the top N items in the product backlog to push into the release backlog as the scope of the release. The schedule can be projected by considering the estimated development effort of the chosen scope and the historical rate of production of the development team.

If the project is schedule-driven, the release schedule must first be determined before the scope can be determined.

Schedule-Driven as the Normal Case

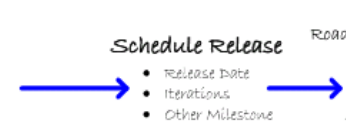
Everyone remembers a schedule slip, but almost no one remembers a scope slip.

Delivering a system with slightly less functionality on time is almost always better than delivering a system with slightly more functionality but late.

For this reason, most agile and iterative development projects are schedule-driven and more typically will adjust scope than schedule.

The list of features, defects and other work are always prioritized by the value to the business. Thus, the items having the lowest priority have the least value and are, by definition, at risk of not being completed during the release (or iteration).

Scheduling the Release



In an iterative development process, the project schedule has two major components. First, there are one or more iterations having known durations and end dates. Second, there is at least one milestone representing the date when the release will be delivered. Most release schedules contain a variety of other events and milestones as well.

Since iterative development processes usually employ time boxing, a large part of scheduling a release is deciding how many iterations there will be and when they will start and end. While it is not necessary for all iterations to have the same duration, it is standard practice since this helps establish the predictable and desirable rhythm that synchronizes the entire development organization.

Fixed iterations automatically define the development schedule: "If it's the last Thursday, we should be planning the next iteration."

The schedule must also take into account events that affect the availability of developers during the release. This might include training sessions, holidays, vacations, trade shows and the like.

In schedule-driven projects, the delivery milestone anchors the release plan and from this date the scope of the project can be estimated. The

available budget of development resources over the course of the release must first be computed. This analysis takes into account the historical production rate of the development team as well as the expected availability schedules of the developers. The historical production rate tells us how much work the development team can do in a given period of calendar time. Then, the highest ranking product backlog items, the sum of whose estimates are less than or equal to the budget of development resources, are selected as the scope of the release. The lowest ranked items in the release backlog are, by definition, at risk of not being completed.

Coping with Uncertainty in Release Planning

Whether a project is scope-driven or schedule-driven, there will be uncertainty associated with the project plan.

- The team's estimates will never be 100% accurate.
- The team's future rate of production cannot be precisely predicted
- There may be significant technological uncertainty associated with new features
- New requirements, defects and other work may be added to the backlog.

It is sensible to make some allowance for this uncertainty. But how can we do that? Generally, we must provide some amount of buffer to the scope, schedule or resources.

For most companies, it is prohibitive to create a resource buffer by employing more developers. It is more likely that the available development resources are fixed, and short iterations make this a virtual certainty.

One buffering approach would add a buffer to the estimate for each item in the backlog. This approach is based on the assumption that everything will go wrong. A lot may go wrong, but it's doubtful that everything will go wrong. Adding up all of the individually buffered estimates will unnecessarily extend the schedule or reduce the scope. This approach will tend to be overly conservative and unnecessarily limit the productivity of the team.

The recommended approach is to apply a modest buffer to the aggregate of the items in the backlog. Anderson summarizes the problem way:

"Planning should be done at the project level and...the project delivery date should be protected by a project buffer determined by the aggregate uncertainty for the project."
(Anderson 2003, p. 71)

Iteration Planning and Execution

Overview

As Figure 9 illustrates, an iteration consists of three major phases. The first is a *planning* phase during which the “iteration backlog” is defined. The second is the *development* phase where the backlog items are developed. The final phase involves *delivery* of the new system increment built during the iteration and *assessment* of the iteration.

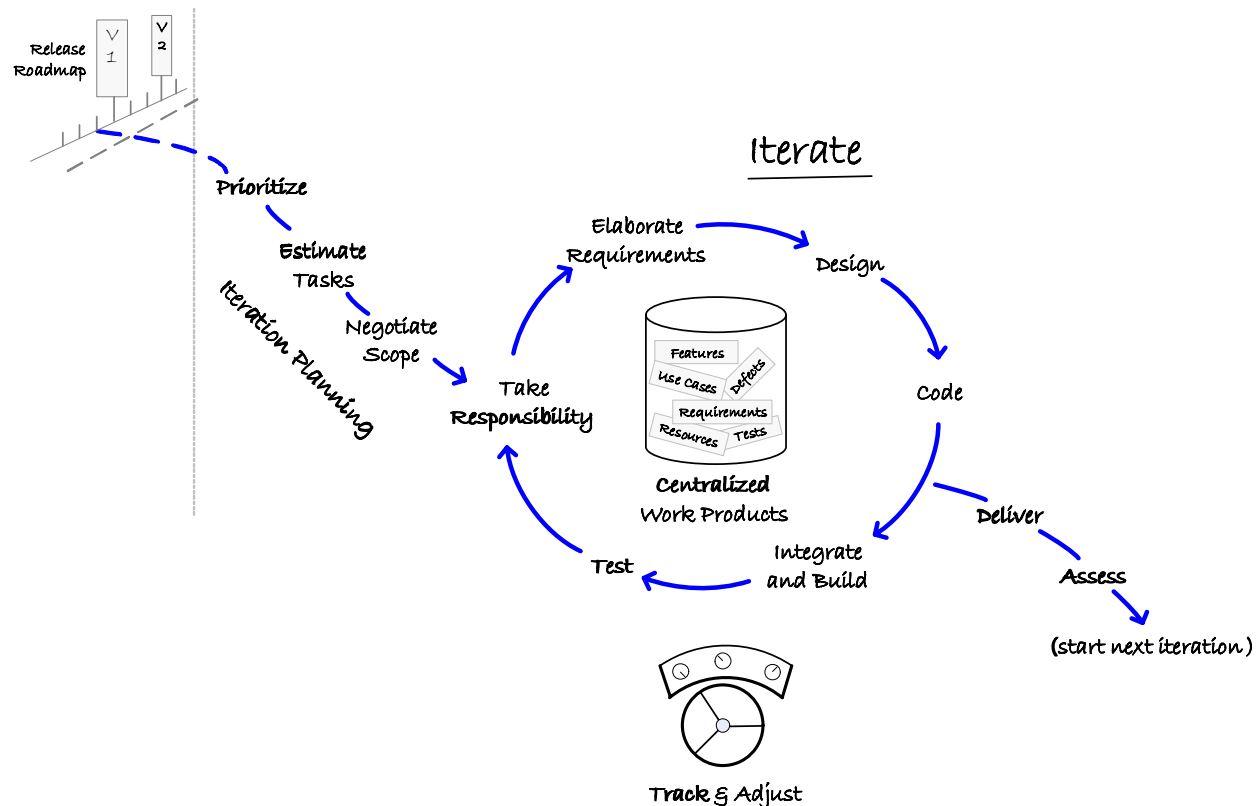
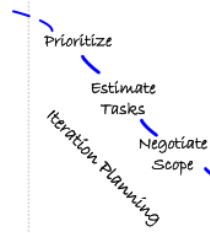


Figure 9 - Iteration Process Model



Iteration Planning

Iterations begin with a short planning period where product owners and developers meet to negotiate a plan. This may take hours or days depending on the length of iterations. Due to time boxing, the duration and end date of the iteration is usually fixed, so it is normally unnecessary to negotiate the schedule of the iteration itself. The primary concern of iteration planning is defining the scope of the iteration.

Iteration planning begins with revising and refining the remaining list of potential work in the release backlog. Product owners, customers and the development team may add or reduce features, defects and other infrastructure work based on the current business situation. Business priority, risk and rough estimates are assigned to new items or may be revised for existing items. The product owners will then re-rank the work items and select a scope of work to propose for the iteration.

Next, an iteration planning meeting is held where the product owners present their proposed scope of work to the development team. The development team is given an opportunity to discuss the proposed work with product owners until each item is well enough understood for the development team to prepare a list of engineering tasks and provide detailed estimates.

The development team will then define and estimate the engineering tasks for each proposed backlog item. The development team will then present their estimates to the product owners.

By simply adding up the development team's estimates, we can calculate the *apparent* scope of the iteration. However, the *final* scope of the iteration is the result of a negotiation between product owners and the development team. During this negotiation, product owners may adjust certain backlog items in ways that make them less costly to develop, trade out entire backlog items for others, or may ask for adjustments to certain estimates provided by development.

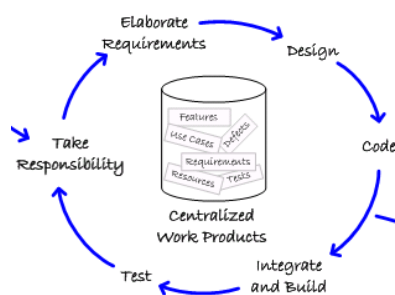
Mary and Tom Poppendieck point out the benefits of iteration planning:

"If you ask a team to choose items from the top of a list that the members believe they can do in a short time-box, the team will probably choose and commit to a reasonable set of features. Once the team members have committed to a set of features that they think they can complete, they will probably figure out how to get those features done within the time-box..."

Good iteration planning gives customers a way to ask for features that are important to them and creates a motivating environment for the development team...As customers see the features they regard as highest priority actually implemented in code, they start to believe the system is going to be real and begin to envision what it can do for them...At the same time, developers gain a sense of accomplishment, and...are even more motivated to satisfy the customers." (Poppendieck 2003, p. 30)

At the end of the iteration planning meeting the product owners and development team jointly *commit* to the iteration plan. After this point, it is usually a rule that only the development team can change the scope of the iteration. Product owners must wait until the next iteration to change the direction of the development effort.

Iteration Execution



Having committed to the iteration plan, the team starts development. Each developer, or pair of developers in organizations doing pair programming, will follow the same basic process repeatedly throughout the iteration until there is no more work in the backlog:

- *Take responsibility* for a backlog item (i.e., feature, use case, defect fix, other task)
- *Develop* (design, code, integrate and test) the backlog item
- *Deliver* the backlog item by integrating it into a system build
- *Declare* the backlog item as developed, signaling that it is ready for acceptance testing

This cycle repeats within an iteration as each developer ultimately takes responsibility for all the backlog items in their queue.

In most organizations developers will also support the management of the process by:

- Participating in the *daily team meeting*
- *Estimating actual and remaining effort* for the backlog items that they are responsible for

Take Responsibility

Having committed to the iteration plan, the team is faced with the question of how to allocate that work to the members of the development team.

"It must be clear to every person, at all times, what he or she should do to make the most effective contribution to the business. When people do not know what to do, time is lost, productivity suffers, and rapid deliver is not possible." [Poppendieck 2003]

The two basic approaches to allocating work are that a manager can assign work to developers, or the developers can choose the work they will do.

As Poppendieck notes, the latter approach is recommended:

“When things are happening quickly, there is not enough time for information to travel up the chain of command and then come back down as directives.”

The process of developers taking responsibility for work must be supported by a visible indicator showing who is responsible for what work, and by daily status meetings where status and issues can be discussed.

Develop

Once a developer takes responsibility for a backlog item, they then:

- Elaborate requirements
- Design
- Program (and refactor as necessary) tests and code
- Execute the test suite on the build
- Integrate program code and tests into a build of the system

During a typical development iteration, the developer will cycle through most of these activities many times. The order in which the above activities occur is a matter of the developer’s programming practices and the particular situation. The developer continues the activities until his or her tests for the new functionality or defect pass. At that point, the developer can confidently include the new functionality in a build of the system.

Deliver

The developer delivers the new functionality or defect fix by checking the code into the source control system and including it in a build of the system. The unit test suite, and other appropriate tests are run before the code is checked in to ensure that the changes do not break the build.

Declare Completion

Once the developer has integrated their work into a build of the system, they can declare the backlog item as complete. This signals other members of the development organization that the backlog item is ready. For example, the testing group now knows they can include the new functionality or defect fix in their testing efforts and can start automated tests of various kinds (functional, acceptance, performance, etc.).

Daily Team Meeting

The daily team meeting is a very short meeting where the development team gathers to raise issues, communicate status and generally coordinate with one another.

The Scrum methodology recommends that each participant answer 3 questions:

- What did you do yesterday?
- What will you do between now and the next daily status meeting?
- What is blocking your progress?

In order to keep the meeting short and focused, the resolution of larger issues are deferred until after the meeting.

Deliver and Assess



At the end of each iteration, the development team delivers a new version of the system to product owners and possibly to external customers.

Delivery is not just a matter of *saying* that the functionality built during the iteration is complete. Rather, it is a matter of *demonstrating* its completion. The team does this by providing access to a running version of the application so that stakeholders can use and evaluate the new version.

The final activity in an iteration is to reflect and assess the results. The goal assessment is to mine the lessons learned during the iteration and then adapt the development process accordingly. This allows the team to continually improve the throughput of the development process and the quality of the resulting system.

The other major activity that occurs during assessment is a “closing” process where unfinished items are put back into the release backlog as work to be done. The iteration is concluded and the iteration backlog becomes a record of the work completed during the iteration.

Small Releases

The culmination of the release cycle is to release a new version of the system to customers.

Developing with an agile and iterative development process opens the door to the possibility of smaller and more frequent releases. Two primary benefits of this change are increased responsiveness and reduced risk.

Responsiveness increases because newly discovered customer needs can be addressed in a much shorter timeframe.

Risk is reduced because customers have the ability to provide feedback as to whether the product is on track earlier in the development process.

Tracking and Adjusting



Track & Adjust

Underlying the development process is the continuous process of tracking status and adjusting course. Even within the course of a short iteration scope must be a managed and deviations from plan addressed. This activity is focused on getting an objective, real time picture of where the software development effort is and how fast it is moving.

Tracking progress of the current iteration requires having visibility into the status of the features, defects and other tasks that are being worked on during the iteration. In particular, it's important to be able to understand how quickly the team is moving through the scheduled work and how accurate their estimates were.

The progress towards the release can be understood by considering the status of the features, defects and other tasks across the iterations in the release. Iterative and incremental processes tend to favor a schedule-driven approach, so at the release level it is most important to understand which chunks of planned work are done and how fast the team is producing work. This information allows the team to deliver the most valuable software on the committed release date by making decisions about what work to do next and what work to defer.

User Feedback

The primary mechanism that allows a team to steer towards its release goals is demonstrating working software early and often to product owners, customers, and hopefully to end users.

It is a reality of software development that customers' understanding of their requirements for a software system tend to evolve as they see and use the software itself.

Thus, every iteration is an opportunity for the team to get feedback and guidance from customers about how to make the system delivered on the release date the most valuable that it can be to the customer.

Throughput and Burn Charts

Since iterations are fixed in duration, the primary way the team and its managers can gauge progress is to continuously monitor current status and also estimate how much work is remaining.

Computing the estimated remaining work in the iteration at a given point in time requires two pieces of information: the total of the estimates for all backlog items that are not yet started and the estimated remaining effort for any in-progress backlog item. The sum of those two amounts represents the estimated remaining work to be completed during the iteration.

Plotting this value each day of the iteration produces what is referred to as a “burndown chart”.

Small Chunks of Work

The size of the chunks of work that are scheduled into the iteration have a dramatic effect on the visibility of their status.

If one large chunk of work is scheduled into an iteration then by implication the entire iteration will be devoted to working towards finishing that one chunk of work and we don't really expect that chunk of work to be completed until the end of the iteration. Gauging the progress of the iteration is a matter of guessing the “percentage complete” of the one monolithic chunk of work. It is also difficult to get the testing group involved early in the iteration when all of the work is delivered in one or a few very large chunks at the end of the iteration.

If we instead break the monolithic chunk of work into smaller pieces, we can then consider the status of the smaller pieces individually and we no longer have to wait until the end of the iteration to see a big chunk of work transition from a state of “in progress” to “complete”. Instead, all along the way, smaller chunks of work are transitioning from “planned” to “in progress” to “complete” state as developers take responsibility for, work on and then deliver each small chunk.

This gives us a more fine-grained means to track the progress of the iteration. We suddenly can understand whether any particular chunk of work is blocked, ready to be tested, completed ahead of schedule or perhaps unlikely to be worked on at all. We also have the ability to deliver the smaller pieces to the testing group sooner during the iteration.

Test Metrics

We discussed earlier that the development team only gets credit for delivering working software accompanied by tests that verify its quality and adherence to requirements.

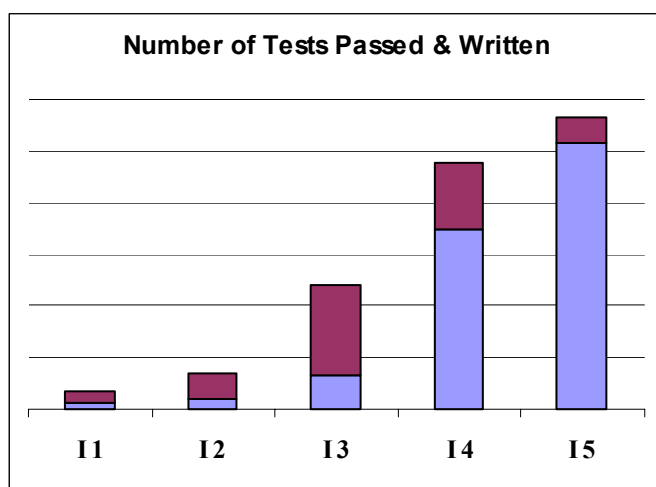


Figure 10 - Tests Written and Passing Graph

Simply delivering code, unaccompanied by tests, is insufficient so all features must have an appropriate set of tests.

As such, another way to gain visibility into the progress of an iteration is to track how many tests are written and how many of those are passing.

The chart in Figure 10 is useful for communicating this kind of information. We would expect to see both the number of tests written and the proportion of tests passing climb over time. It is a red flag if the trend line of tests written starts to flatten or if the proportion of tests passing remains constant or drops.

Issue List

Another source of information about the status and progress of an iteration is the issue list. This list is often owned by the engineering manager, but

issues may be contributed by any team member. Often issues are raised verbally during the daily meeting. Issues may relate to unmet infrastructural or technical needs of the development team, deficiencies in requirements, or disagreements about how to proceed, to name a few.

No matter their nature or source, issues usually represent threats to the team's ability to deliver on its commitment and thus are an important source of information about the development process.

Conclusion

The intensely competitive nature of the software product marketplace continues to put more pressure on today's software product teams. Moreover, as the market matures, the scope and sophistication of the applications that must be deployed to address user needs is growing dramatically. Team size is increasing while development times decrease. Agile software development methods address these challenges by providing teams with a way to deliver smaller increments of functionality to the customer more quickly than before. In turn, this accomplishes three business objectives:

- 1) Faster time-to-market and subsequent larger market share
- 2) Better fit of software features to customer's true needs for increased product preference
- 3) Adapt to changing business conditions faster than your rivals

At the core of all these agile methods is iterative and incremental development, a technique that provides objective feedback on a real time basis. With iterations, teams have *visibility* into the product under development, and they have more time to adapt and adjust to changing customer needs.

Adopting these new practices fundamentally changes the way teams plan, run and manage software projects. Moreover, scaling these agile methods to increasingly demanding applications requires adaptive requirements, test and project management practices as well as appropriate tooling to support larger and more distributed teams.

Mastering these practices gives teams a true competitive edge, and edge that allows them to better meet both their personal and business objectives. These are the teams that will be the most competitive in the future, and these teams will ***deliver early and deliver often***.

Suggested Reading

Larman, Craig. 2004. *Agile & Iterative Development – A Manager’s Guide*. Boston, MA: Addison-Wesley.

Leffingwell, Dean, Don Widrig. 2003. *Managing Software Requirements, Second Edition*. Boston, MA: Addison-Wesley.

Schwaber, Ken, Mike Beedle. 2002. *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall.

Bibliography

Anderson, David J. 2003. *Agile Management for Software Engineering – Applying the Theory of Constraints for Business Results*. Upper Saddle River, NJ: Prentice Hall.

Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley.

Kruchten, Phillippe. 2001. “Agility with the RUP.” *Cutter IT journal* 14(12) p. 27.

Kruchten, Phillippe. 2004. *The Rational Unified Process: An Introduction, Third Edition*. Boston, MA: Addison-Wesley.

Larman, Craig. 2004. *Agile & Iterative Development – A Manager’s Guide*. Boston, MA: Addison-Wesley.

Leffingwell, Dean, Don Widrig. 2003. *Managing Software Requirements, Second Edition*. Boston, MA: Addison-Wesley.

Poppendieck, Mary, Tom Poppendieck. 2003. *Lean Software Development: An Agile Toolkit*. Boston, MA: Addison-Wesley.

Rational Software Corporation. 2002. “Rational Unified Process 2002.” Cupertino, CA: Rational Software Corporation.

Schwaber, Ken, Mike Beedle. 2002. *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall.